AS_STAR

Matthew Taliaferro

6/22/2020

Overview

Last week we talked about quantifying gene expression with `salmon` and identifying differentially expressed genes with `DESeq2`. This week, we are going to talk about analyzing the regulation of alternative splicing using RNAseq approaches. As we learned last week, `salmon` quantifies a fastq file of sequencing reads against a fasta file of all transcripts present in the sample. At the end of this analysis, we end up with quantifications for each transcript in our fasta file. However, for splicing you may be able to see how this strategy may need to be tweaked. `salmon` gave us transcript-level data, but for looking at splicing, we often want to measure how the inclusion of individual **exons** within transcripts differs between conditions. Obviously, transcript-level quantifications are not that useful here.

> Small aside: Actually, transcript level quantifications could work, because you could ask how the relative abundances of two different transcripts (one that has the exon in question and one that doesn't) vary across conditions. See also `suppa2`.

So what we need here is exon-level quantifications. One way to do that is to literally count reads that either support the inclusion of an exon or support its exlcusion.

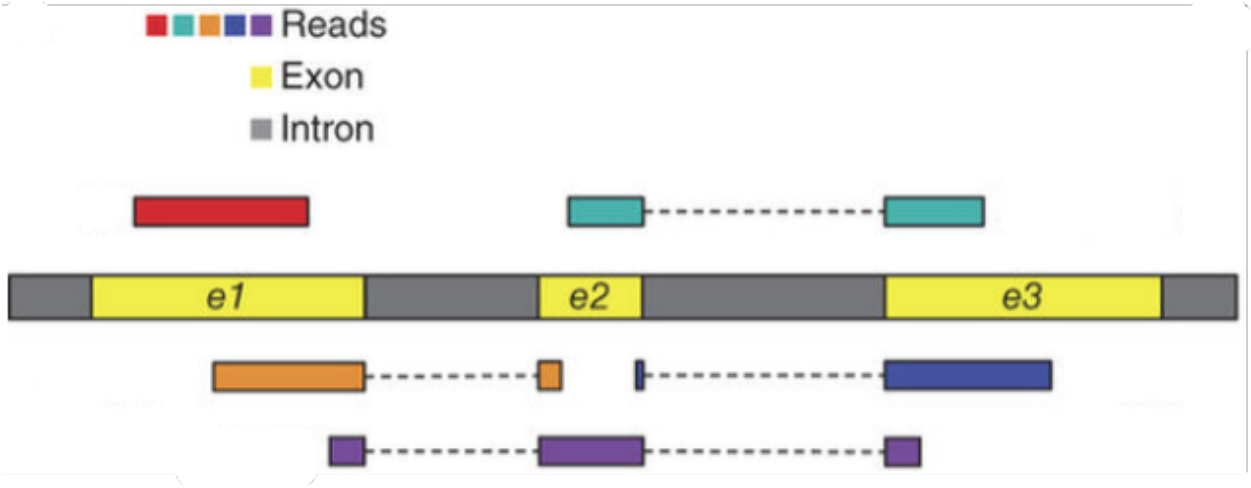


Figure 1: Kim et al, Nat Methods, 2015

Here we can see examples of some RNAseq reads mapped along a transcript. This transcript contains exons (yellow) and introns (gray). Let's say that there are two isoforms of this gene: one where exon2 is included and one where it is exlcuded. Reads have been "aligned" to this transcript to give a graphical representation of where they came from. You can see that the orange, purple, blue, and teal reads all *support* the inclusion of exon2. Another way to think about this is that these reads came from RNA molecules in which the transcript was included.

How do we know this? Well each of these reads cross a **splice junction** that is either exon1-exon2 or exon2-exon3. These reads tell us, unambiguously, that exon2 was included in the RNA molecule that these reads came from.

> What does the red read tell us? What would a read that unambiguously told us that exon2 was *excluded* look like?

Here we can see an example of this strategy in action.

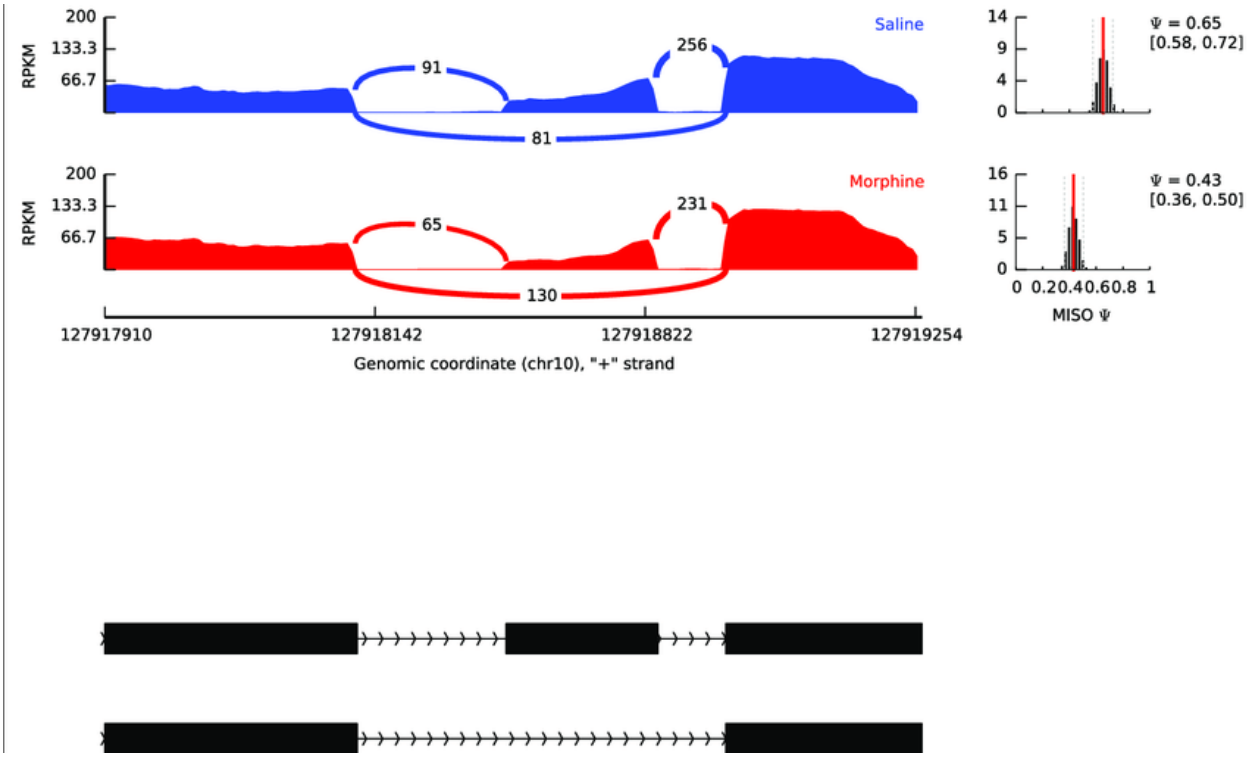


Figure 2: Ye et al, Diagnostic Pathology, 2014

This is an example of reads mapped to the area surrounding an alternative exon (the middle exon). The height of the red and blue area corresponds to the number of reads that cover that location. The red and blue lines connecting exons represent the number of reads that span that junction. So for the blue condition, there are 347 reads (91 + 256) that support the inclusion of this exon, while 81 reads support its exclusion. In the red condition, this exon is less often excluded as 296 reads (65 + 231) support its inclusion while 130 support its exclusion. You can think about how often an exon was included in a sample as a competition between the inclusion- and exclusion-supporting reads.

So you can see that to use this approach, we need to know *where* in the transcript these reads came from. To do this, we need to align the reads to the genome (or the transcriptome I guess, to be precise). You have probably already done something similar to this in the DNA block using `bowtie`. Sequence aligners, like `bowtie` will take a sequence from a fastq file, search a genome or set of reference sequences, and return the coordinates for the best match of the query and reference sequences.

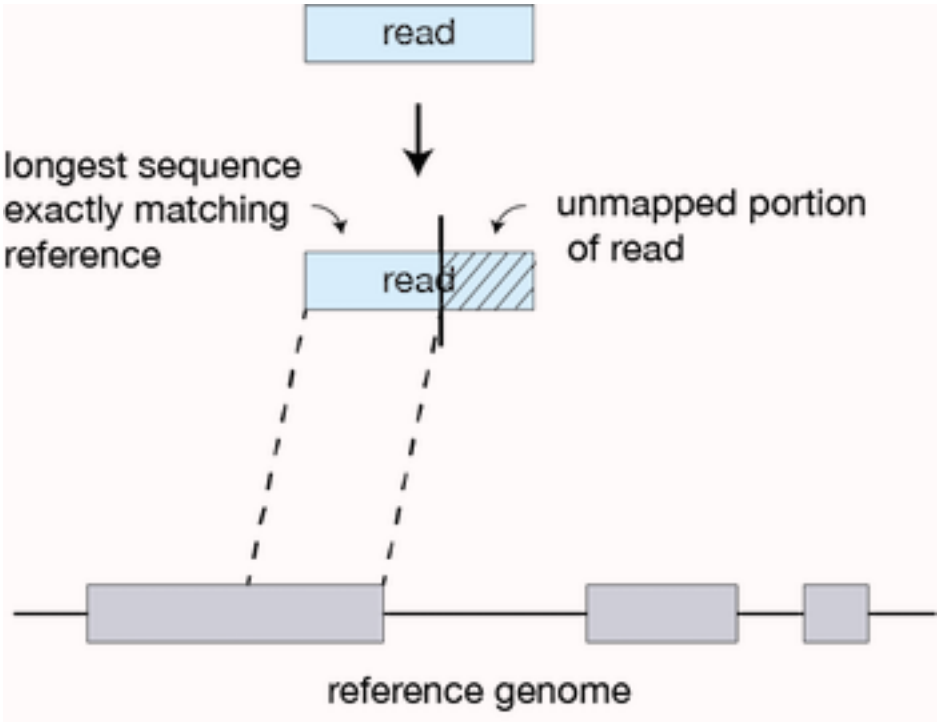
However, as you might guess, there's a hiccup here when we start to talk about aligning reads that came from RNA. The problem is introns. With DNA sequences, such as those you saw obtained from a ChIP-seq experiment, queries are completely contigouous with respect to the reference. This makes the aligner's job relatively easy. But with RNA, you could have a query that matches the reference for a bit, then jumps (over the intron) and recommences matching the reference. We will begin now with writing our own algorithm and implementing it to perform these alignments in an efficient manner.

(kidding)

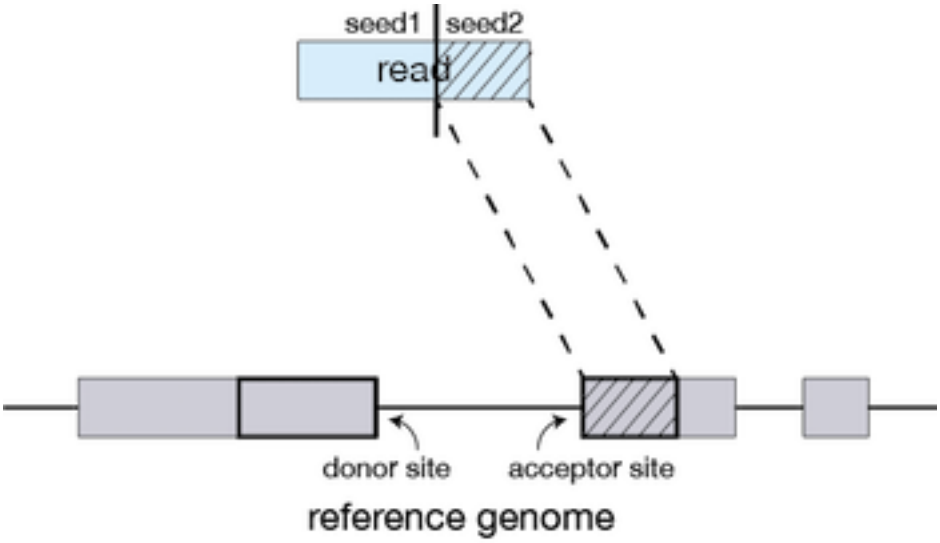
There are multiple tools that will perform these alignments. One of the popular ones, and the one that we will use for this course, is called `STAR`.

How STAR works

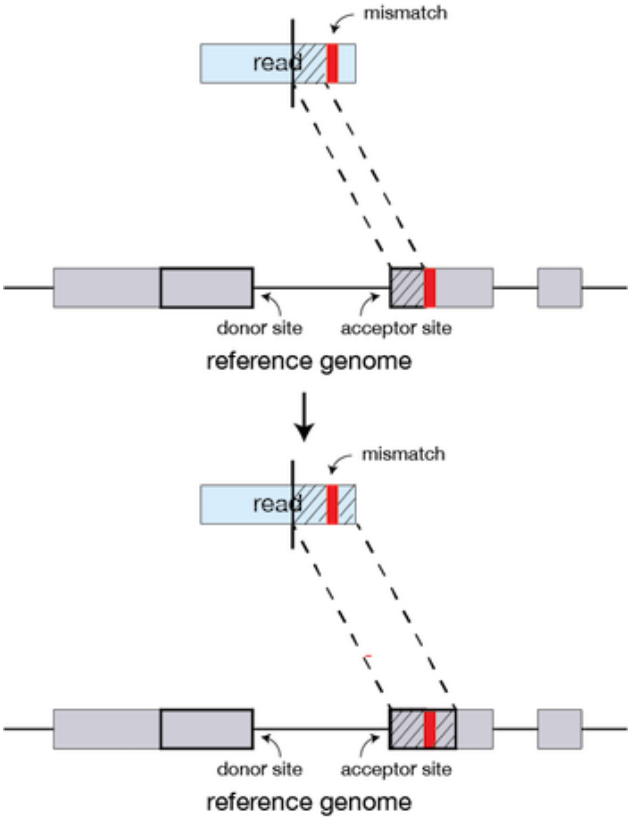
STAR begins by finding matches (either unique or nonunique) between a portion of a read and the reference. This matching region of the query is extended along the reference until the two start to disagree. If this match extends all the way through to the end of the read, then the read lies completely within one exon (or intron, or I guess intergenic region if you are bad at making RNAseq libraries) and we are done. If the match ends before the end of the read, the part that has matched so far defines one *seed*.



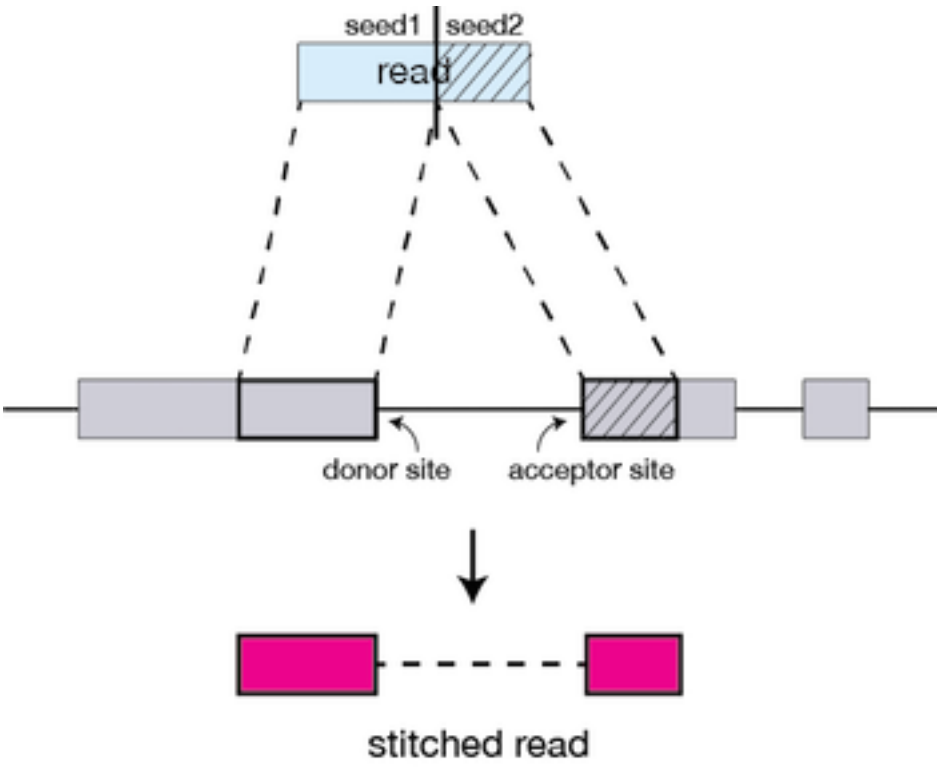
STAR then takes the rest of the query and uses it to find the best match to its sequence in the reference, defining another seed.



If, during the extension of a match a small region of mismatch or discontinuity occurs, these can be identified as mutations or indels if high-quality matches between the query and reference resume later in the read.



After aligning seeds, they can be stitched together. The stitching of seeds with high alignment quality (low number of indels, mismatches) is prefered over the stitching of seeds with low alignment quality (high number of indels, mismatches).



Running STAR

To align reads, we first need to create an **index** of the genome (see STAR manual here. To do this, `STAR` will require the sequence of the genome (in fasta format), and an annotation that tells it where exons and introns are. It needs the annotation to be able to see if seeds that it stitches together make sense with what we know about exon/intron structures that exist in the transcriptome. Let's take a look at one of these genome annotation files.

Annotation files

Genome annotation files come in a few flavors, but the one we are going to use is called **gff**. Here's an example of part of one.

```
head -n 4 ../data/dummySTAR/MOLB7950.gff3
```

```
## ##sequence-region chr19 1 61431566
## chr19    HAVANA  gene    3065711 3197714 .   -   .   ID=ENSMUSG00000100969.6;gene_id=ENSMUSG00000100
## chr19    HAVANA  transcript  3065711 3197714 .   -   .   ID=ENSMUST00000190575.6;Parent=ENSMUSG00000
## chr19    HAVANA  exon    3197605 3197714 .   -   .   ID=exon:ENSMUST00000190575.6:1;Parent=ENSMUST00
```

We see here three lines (ignore the first one that starts with ##). Each line corresponds to one feature. This is a tab-delimited text file. There are only a few columns that we care about:

- Column 1: chromosome
- Column 3: feature type
- Column 4: feature start
- Column 5: feature end
- Column 7: strand (you aren't in DNA land anymore... strand matters)

Column 8 contains various information about the feature. Perhaps the most important one tells you about the hierarchy that defines the relationship between features. For example, genes contain *children* transcripts

within them, and each transcript contains *children* exons. Transcripts will therefore belong to *parent* genes and exons will belong to *parent* transcripts. Biologically, this should make sense to you. These relationships are indicated by the **Parent** attribute within column 8.

Make STAR index

OK now we are ready to make our index. There relevant options we will need to pay attention to when doing this are shown below:

- –**runMode** genomeGenerate (we are making an index, not aligning reads)
- –**genomeDir** /path/to/genomeDir (where you want STAR to put this index we are making)
- –**genomeFastaFiles** /path/to/genomesequence (genome sequence as fasta, either one file or multiple)
- –**sjdbGTFfile** /path/to/annotations.gff (yes it says gtf, but we are going to use a gff format)
- –**sjdbOverhang** 100 (100 will usually be a good value here, the recommended value is readLength - 1)
- –**sjdbGTFtagExonParentTranscript** Parent (we have to specify this because we are using a gff annotation and this is how gff files denote relationships)
- –**genomeSAindexNbases** 11 (don't worry about this one, we are specifying it because we are using an artifically small genome in this example)

Now we are ready to make our index. This should take a couple minutes.

```
source activate three

STAR --runMode genomeGenerate --genomeDir ../data/dummySTAR/dummySTARindex --genomeFastaFiles ../data/du
```

```
## Sep 17 14:58:32 ..... started STAR run
## Sep 17 14:58:32 ... starting to generate Genome files
## Sep 17 14:58:33 ..... processing annotations GTF
## Sep 17 14:58:33 ... starting to sort Suffix Array. This may take a long time...
## Sep 17 14:58:34 ... sorting Suffix Array chunks and saving them to disk...
## Sep 17 14:59:44 ... loading chunks from disk, packing SA...
## Sep 17 14:59:45 ... finished generating suffix array
## Sep 17 14:59:45 ... generating Suffix Array index
## Sep 17 14:59:48 ... completed Suffix Array index
## Sep 17 14:59:48 ..... inserting junctions into the genome indices
## Sep 17 14:59:56 ... writing Genome to disk ...
## Sep 17 14:59:56 ... writing Suffix Array to disk ...
## Sep 17 14:59:56 ... writing SAindex to disk
## Sep 17 14:59:57 ..... finished successfully
```

Align reads

Now that we have our index we are ready to align our reads. The options we need to pay attention to here are:

- –**runMode** alignReads (we are aligning this time)
- –**genomeDir** /path/to/genomeDir (a path to the index we made in the previous step)
- –**readFilesIn** /path/to/forwardreads /path/to/reversereads (paths to our fastqs, separated by a space)
- –**readFilesCommand** gunzip -c (our reads our gzipped so we need to tell STAR how to read them)
- –**outFileNamePrefix** path/to/outputdir (where to put the results)
- –**outSAMtype** BAM SortedByCoordinate (the format of the alignment output, more on this later)

Now we are ready to align our reads.

```
source activate three

STAR --runMode alignReads --genomeDir ../data/dummySTAR/dummySTARindex/ --readFilesIn ../data/dummySTAR/
```

```
## Sep 17 14:59:57 ..... started STAR run
## Sep 17 14:59:57 ..... loading genome
## Sep 17 14:59:57 ..... started mapping
## Sep 17 15:00:25 ..... finished mapping
## Sep 17 15:00:25 ..... started sorting BAM
## Sep 17 15:00:26 ..... finished successfully
```

Monitoring mapping stats produced by STAR

Well so how did it go? We can look at the log file to find out.

```
less ../data/dummySTAR/MOLB7950/dummyLog.final.out
```

```
##                                 Started job on |    Sep 17 14:59:57
##                              Started mapping on |    Sep 17 14:59:57
##                                     Finished on |    Sep 17 15:00:26
##        Mapping speed, Million of reads per hour |    12.41
##
##                            Number of input reads |    99997
##                        Average input read length |    288
##                                     UNIQUE READS:
##                    Uniquely mapped reads number |    96790
##                         Uniquely mapped reads % |    96.79%
##                           Average mapped length |    274.37
##                        Number of splices: Total |    95609
##             Number of splices: Annotated (sjdb) |    94474
##                        Number of splices: GT/AG |    95200
##                        Number of splices: GC/AG |    277
##                        Number of splices: AT/AC |    0
##                Number of splices: Non-canonical |    132
##                       Mismatch rate per base, % |    0.27%
##                          Deletion rate per base |    0.01%
##                         Deletion average length |    2.10
##                         Insertion rate per base |    0.01%
##                        Insertion average length |    1.67
##                              MULTI-MAPPING READS:
##         Number of reads mapped to multiple loci |    1031
##              % of reads mapped to multiple loci |    1.03%
##         Number of reads mapped to too many loci |    11
##              % of reads mapped to too many loci |    0.01%
##                                   UNMAPPED READS:
##   Number of reads unmapped: too many mismatches |    0
##        % of reads unmapped: too many mismatches |    0.00%
##             Number of reads unmapped: too short |    2143
##                  % of reads unmapped: too short |    2.14%
##                Number of reads unmapped: other |    22
##                     % of reads unmapped: other |    0.02%
##                                   CHIMERIC READS:
##                         Number of chimeric reads |    0
##                              % of chimeric reads |    0.00%
```

We can see that we put in almost 100k reads (or more accurately read pairs), and 96.7k of these could be uniquely assigned to a single genomic position. 95.6k of these had a splice junction. This is expected for paired end reads against a genome with many introns and short exons, like the mouse (or any mammalian) genome.

As an aside, any read that aligns more times than is allowed by the flag –**outFilterMultimapNmax** is not reported in the alignments. As a default, this value is set to 10. Libraries that are made from low complexity RNA samples and those that deal with repetitive genomic regions can be sensitive to this. Also, if you wanted to, you can use this flag to restrict your alignment file to those that only *uniquely* aligned by setting this value to 1.

Investigating alignment files

Our alignment output file is `dummyAligned.sortedByCoord.out.bam`. Lets take a look at it to see what's going on. Well, we could, if you read binary. This is a binary form of a different alignment file format called SAM. Luckily, we can easily convert between the binary BAM format and the plain text SAM format with `samtools`.

`samtools view` will allow us to convert our BAM file into a SAM file.

```
samtools view ../data/dummySTAR/MOLB7950/dummyAligned.sortedByCoord.out.bam > ../data/dummySTAR/MOLB795
```

SAM files can be a little confusing, but it's worth taking the time to get to know them. The full SAM format specification can be found here.

Let's take a look at our SAM file and see what we see. I'm going to pick 2 lines out.

```
grep A00405:98:HK5KVDSXX:4:1653:27489:15186 ../data/dummySTAR/MOLB7950/dummyAligned.sam
```

```
## A00405:98:HK5KVDSXX:4:1653:27489:15186   163 chr19   3371611 255 67M3415N84M =   3371629 3584    CAG
## A00405:98:HK5KVDSXX:4:1653:27489:15186   83  chr19   3371629 255 49M3415N102M    =   3371611 -3584
```

Here we are looking at 2 lines from this file. These two lines correspond to two paired reads. I know that because the first field in this file is the read ID as it came off the sequencer. You can see that these two reads have the same ID (it's the thing I grepped for).

The **second** field is a bitwise flag. It is a sum of integers where each integer tells us something about the read. Every possible value of this flag is a unique combination of the informative integers. You can see what each of these integers are and what they mean in the SAM format specification. There is also a handy calculator that you can plug your value into and it will tell you what your flag means here. If we put our first flag, 163, in there it tell us that this read is:

- The second read in a mate pair (128)
- On the opposite strand of its mate pair (32)
- Is mapped and properly paired (2)
- Is paired (1)

If you put the flag value for the second read into the calculator, what do you get?

The **third** field is obviously the reference name. No big mystery there. This read maps to chromosome 19.

The **fourth** field is the position on the reference that corresponds to the beginning of the query. This read starts to map to chr19 beginning at position 3371611. As a aside, positions reported in SAM files are 1-based, not 0-based.

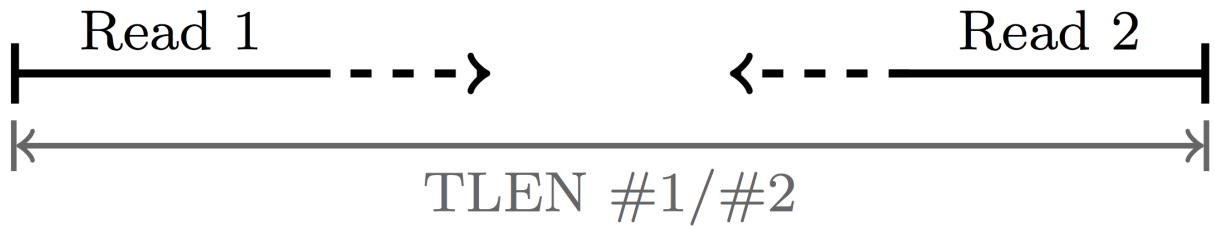
The **sixth** field is called the CIGAR string. This is a string of characters that tells you a little bit about *how* the query aligns to the reference. Again, details can be found in the SAM format specification. The CIGAR string for the first read can be interpreted as follows:

- The first 67 bases in the query align to the reference.
- There is then a gap in the reference of 3415 nt.
- Then the query starts to match again, and does so for the next 84 nt.

These are paired end 151 nt reads, so it makes sense that 67 + 84 = 151.

In not so shocking news, the top read's mate (the second read) also has a gap in the reference of 3415 nt. As you might have guessed, these reads are spanning the same intron, which you would expect reads from the same mate pair to do.

The **ninth** field is the *template length*, abbreviated TLEN. This is the distance, in nt, from the beginning of one read in a pair to the end of it's mate.



If you know a little bit about how RNAseq libraries are made, you might know that transcripts are fragmented, usually to lengths of 200-500 nt. Given that this read is stretching over 3 kb along the reference sequence, it's a good bet that it is spanning an intron that is present in the reference but had been removed in the RNA molecule.

> Some images in this lesson were created by members of the teaching team at the Harvard Chan Bioinformatics Core.